LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-659977

A Global Exception Fault Tolerance Model for MPI

I. Laguna, T. Gamblin, K. Mohror, M. Schulz, H. Pritchard, N. Davis

September 8, 2014

ExaMPI
New Orleans, LA, United States
November 16, 2014 through November 16, 2014

Disclaimer

A Global Exception Fault Tolerance Model for MPI

Ignacio Laguna[†], Todd Gamblin[†], Kathryn Mohror[†], Martin Schulz[†], Howard Pritchard[*], and Nickolas Davis[‡]
[†]Lawrence Livermore National Laboratory
[*]Los Alamos National Laboratory
[‡]New Mexico Institute of Mining and Technology

I. Introduction

Driven both by the anticipated hardware reliability constraints for exascale systems, and the desire to use MPI in a broader application space, there is an ongoing effort to incorporate fault tolerance constructs into MPI. Several fault-tolerant models have been proposed for MPI [1], [2], [3], [4]. However, despite these attempts, and over six years of effort by the MPI Forum's [5] Fault Tolerance (FT) working group, the limited success to date to introduce fault tolerance constructs into MPI reflects the complexity of the problem. This complexity stems in part from the fact that MPI incorporates many concepts besides simple point-to-point communication protocols, such as collectives, communicators, message ordering and wildcard receives, collective I/O operations, and one-sided operations. Concepts that MPI lacks (e.g., connections and timeouts) also complicate the problem. In addition to the challenges of introducing implementable fault tolerance constructs, there is the question of usability and applicability of such constructs in existing production HPC applications. For example, a **run-through** or forward recovery model, in which the application attempts to find a new state (not necessarily saved previously) from which it can continue operation, may not be as usable for typical bulk synchronous HPC applications as a **roll-back** recovery model, in which the application is restarted from a previously saved state. Given the wide range of potential use cases for MPI fault tolerance, it is likely that a number of approaches will need to be explored.

Recently, the MPI Forum FT working group's efforts have coalesced around the User Level Fault Mitigation (ULFM) proposal [4]. The proposal provides an interface for an application to continue using MPI under fail-stop scenarios. MPI is responsible for reporting process failures to the application, but the application is responsible, via new MPI functions, to bring MPI back to a state where it can continue to be used. ULFM does not provide for an explicit means to restart failed processes. Several researchers have investigated using ULFM both for computational kernels [6] and production applications [7]. These researchers encountered several significant shortcomings of ULFM that will likely either limit its use to a subset of HPC applications, or else require significant, potentially complex enhancements to ULFM to support restarting of failed ranks to provide global backward failure recovery.

The results of these studies, as well as reliability requirements for a number of key HPC applications in the exascale time frame, indicate the need to continue pursuing alternative MPI fault tolerance models. Additionally, it is anticipated that technology trends within the exascale time frame, such as additional options for persistent storage media (e.g., non-volatile memory (NVM) and low cost SSDs), will have a significant impact on fault tolerance approaches in HPC. For example, the availability of NVM on compute nodes of an exascale system will allow for low cost checktpointing and fast roll back of bulk synchronous applications in the event of a process failure—there would typically be no need to reload checkpointed data from distant, slow parallel file systems.

II. Global Exception Recovery Model

With these considerations in mind, we are investigating a different model for MPI fault tolerance: a global-exception, roll-back recovery model. In contrast to ULFM, the basic idea is that upon detecting a fail-stop failure, MPI reinitializes itself—it returns MPI to its state prior to returning from `MPI_Init`, and restarts the application at an application-specified restart point. MPI is also responsible for restarting any failed rank(s). This model implies the presence of a *strongly accurate* (no process is reported as failed till it has actually failed) and *strongly complete* (all surviving elements of the runtime eventually know about the failed process) fault detector within the MPI runtime. This model is not only well suited to a number of important bulk synchronous, production HPC applications, but is also applicable to almost any parallel program model.

A. MPI initialization

The interface defines states to specify under what circumstances a process has been initialized. For example, a process could be in a fresh state (`MPI_START_NEW`), in a re-started state due to a failure (`MPI_START_RESTARTED`), or it has been added to the existing job (`MPI_START_ADDED`). A fault-tolerant MPI program requires calling the `MPI_Reinit` routine to specify a pointer for the main entry point after a failure occurs. Callers of this function should pass command line arguments, and a function to be invoked each time this process restarts. Re-initialization states must be passed according to how the failure occurred and the process started up. A summary of this interface is shown in Figure 1.

```
/***** Initialization routines *****/
typedef enum {
  MPI_START_NEW,          // Fresh process
  MPI_START_RESTARTED,    // Process restarted due to a fault
  MPI_START_ADDED,        // Process is new but added to existingjob
} MPI_Start_state;

// Function pointer type of main entry point
typedef void (*MPI_Restart_point)
        (int argc, char **argv, MPI_Start_state state);

// Marks the start of a resilient MPI program
int MPI_Reinit(int argc, char **argv,
               const MPI_Restart_point point);

/***** Cleanup routines *****/
typedef enum {
  MPI_CLEANUP_ABORT,      // Cleanup failed
  MPI_CLEANUP_SUCCESS,    // Continue rollback
} MPI_Cleanup_code;

// An error handler type that cleans up application or libraryresources
typedef MPI_Cleanup_code
        (*MPI_Cleanup_handler)
        (MPI_Start_state start_state, void *state);

// Functions to push and pop handlers, which are executed inLIFO order
int MPI_Cleanup_handler_push(
    const MPI_Cleanup_handler handler, void *state);
int MPI_Cleanup_handler_pop(
    const MPI_Cleanup_handler *handler, void **state);

/***** Failure notification control *****/
typedef enum {
  MPI_SYNCHRONOUS_FAULTS,
  MPI_ASYNCHRONOUS_FAULTS
} MPI_Fault_mode;

// Get and set fault mode
int MPI_Get_fault_mode(MPI_Fault_mode *mode);
int MPI_Set_fault_mode(MPI_Fault_mode mode);

// Test for faults synchronously
int MPI_Fault_probe();

// Send failure notifications to all processes
int MPI_Fault();
```

Fig. 1. Summary of the MPI global-exception interface.

B. Cleanup handling

Our model incorporates features for use in multi-library (layered) applications, including the notion of cleanup callback functions, and the ability to switch between synchronous and asynchronous process-failure handling. Libraries can **push** any number of cleanup callbacks onto a cleanup function stack—perhaps an initial callback function which handles cleanup of resources allocated when the library (e.g. HDF) was initialized by the application, plus additional cleanup handlers (depending on the call sequence into the library). This is done using the `MPI_Cleanup_handler_push` and `MPI_Cleanup_handler_pop` function shown in Figure 1. A cleanup handler can return any of two states: `ABORT` or `SUCCESS`, which specifies whether the cleanup failed (and the applications ahould abort) or it succeeded (and the application should continue rollback). We intend that pushing and popping cleanup handlers should be treated as a lightweight operation by the MPI implementation.

To work effectively in this model, a library must register callback functions with sufficient roll-back functionality. This involves library's functionalities to roll back the library state to a state that is equivalent its pre-initialization state.

```
#include <mpi.h>

MPI_Cleanup_code cleanup_handler(
                MPI_Start_state start_state, void *s);

// Real main method of the application (entry point forrollbacks)
void resilient_main(
     int argc, char **argv,
     MPI_Start_state start_state)
{
  // Check if the new world size is acceptable. If it is not,abort.
  // Figure out what process died, and receover based on that
  // Load checkpoint if necessary
  // Enter main computation loop (at appropriate step)
  // Store checkpoints
}

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);

  // Register the global app cleanup handler
  MPI_Cleanup_handler_push(cleanup_handler, 0);

  // Init libraries, which could register their own cleanuphandlers
  initialize_libraries(MPI_COMM_WORLD);

  // This is the point at which the resilient MPI programstarts
  MPI_Reinit(argc, argv, resilient_main);

  MPI_Finalize();
}
```

Fig. 2. Sample fault-tolerant program.

C. Synchronous and Asynchronous fault handling

If there are regions of code in the application or library that must be protected from asynchronous reinitialization, the model supports the notion of synchronous failure notification. Analogous to signal masking, an MPI rank can locally set the fault detection mode to synchronous. This prevents MPI from possibly restarting the rank until either it explicitly probes for faults, or it sets the fault detection mode back to asynchronous mode. Using the interface on in Figure 1, the application can set and check the fault mode dynamically. We also provide functionality to test for faults (`MPI_Fault_probe`) and to propagate fault information to alive processes (`MPI_Fault`).

D. Sample fault-tolerant MPI program

Figure 2 shows a sample fault-tolerant MPI program. Note that there is a `main` function, and a `resilient_main` function in which the application computation and failure recovery code is executed. After initializing MPI both the application and libraries push cleanup handlers into the stack, which is followed by a call to the `resilient_main` function. Note that, in contrast to ULFM, there is no need for the application to handle local failure information, such as revoking or shrinking communicators, or determining in what MPI routine the fault occurred (or it manifested on)—all of these is transparently managed by MPI. A more descriptive version of this interface can be found at [8].

III. OpenMPI Prototype

We developed an initial prototype of the proposed model using OpenMPI. The focus of this initial study was ascertaining the level of effort required to perform an MPI *reinitialization* procedure, as well as to get some basic performance measurements comparing the cost of a standard checkpoint/restart procedure to that using a MPI reinitialization approach. Since one of the major difficulties to implementing the reinitialization procedure involves shutdown and restart of network related resources, we targeted three different platforms for initial investigation: a cluster with a Mellanox IB interconnect, using the OpenMPI ibverbs BTL network interface layer; a cluster with a Intel/Qlogic IB interconnect, using the OpenMPI PSM MTL network interface layer; and a Cray XC30 system, using the OpenMPI uGNI BTL network interface layer.

The `MPI_Reinit` method was implemented by selecting the internal steps of the OpenMPI's `MPI_Init` and `MPI_Finalize` procedures required to cleanup resources associated with MPI objects (e.g., constructors, and partially delivered messages). Not surprisingly, we identified and fixed a number of procedures in the `MPI_Finalize` where resources were not being completely released. We also added routines for canceling interrupted messages, and freeing resources associated with them. It turned out that the OpenMPI BTL network interfaces proved to be well suited for the MPI reinitialization procedure, and in and of themselves presented no significant difficulties. Note that, in this prototype, all resources associated with BTL's were torn down and restarted during MPI reinitialization. A more refined approach would likely only do selective cleanup to avoid reconnection costs as a job continued across a reinitialization boundary. Use of the PSM MTL layer proved much more difficult. Owing to time constraints further work using the PSM MTL layer was discontinued in this initial investigation.

Additional work was required in the OpenMPI's runtime layer (ORTE) to implement the reinitialization procedure. The runtime's group communication layer (used for out-of-band data exchange and synchronization) was enhanced to support more general data exchange and synchronization. Before these modifications, the communication pattern for an MPI job startup/shutdown was hard-wired into the ORTE layer.

Using this enhanced OpenMPI/ORTE infrastructure, we modified a highly scalable, molecular dynamics application (ddcMD) [9], [10] as well as a Lattice Bolzmann transport code (LBMv3) [11] to use this prototype. To approximate the asynchronous mode of the proposed method, we modified the main time-step loops of the applications to explicitly execute the `MPI_Reinit` procedure upon receipt of a SIGUSR1 signal. The difference between the time to perform a reinitialization verses a standard job restart with read of the checkpoint file was significant for the LBMv3 code. The reinitialization procedure benefits partially from the fact that at least some portion of the checkpoint data from the last write to the parallel file system is likely still cached in the kernel's buffer cache, since the job was not killed, and the ranks maintain their node locality across the reinitialization boundary. Table I shows a comparison of the reinitialization times and standard job restart times for the LBMv3 on a Cray XC30 system using a strong scaling problem. To simulate the availability of NVM or local SSDs, the LBMv3 was further modified to allow for optional writing of checkpoint files to a local *ramfs* file systems on the compute nodes. The reduction in time for a MPI reinitialization restart verses a full job restart using this approach for storage of checkpoint files was significant for all job sizes tested. Both for the disk and *ramfs* based checkpoint methods, the timing improvements using the MPI reinitialization approach were significant enough to continue pursuing implementation of the proposed Global Exception fault tolerance model in the OpenMPI prototype.

TABLE I
LBMv3 `MPI_Reinit` vs Full restart Time (secs) on a Cray XC30

No. Ranks	job restart	using `MPI_Reinit`
64	42	40
128	22	5.9
200	13	4.7

IV. Ongoing Work and Workshop Presentation

Next steps in evaluating the viability of a Global Exception recovery model include incorporating a strongly accurate and strongly complete fault detector into the ORTE runtime. Such a detector is necessary in order to repair the ORTE internal communication network, as well as to determine which ranks need to be restarted. The ORTE infrastructure for supporting MPI-2 `MPI_Comm_spawn` functionality will be enhanced to restart failed ranks. The remaining elements of the proposed MPI extensions will also be implemented.

We will further modify the ddcMD and LBMv3 applications to make full use of the proposed recovery model, including use of cleanup callbacks, and handling of restarted ranks. The modified applications will be used for testing the practicality of the model in bulk synchronous HPC applications.

We will discuss our experience incorporating the proposed MPI recovery model within the applications, as well as the effectiveness of the model in the face of fail-stop process failures. We will present more detailed analysis of timing comparisons between MPI *reinitialization* restarts and standard full job restart from a previous checkpoint.

Acknowledgment

The authors would like to thank Ralph Castain (Intel) for changes to the Open Runtime (ORTE) to support this effort. The authors also thank Nathan Hjelm (Los Alamos National Lab) for his help with the LBMv3 application.

References

[1] G. E. Fagg and J. J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *Recent advances in parallel virtual machine and message passing interface*. Springer, 2000, pp. 346–353.

[2] J. Hursey, R. L. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. G. Solt, "Run-through stabilization: An MPI proposal for process fault tolerance," in *Recent Advances in the Message Passing Interface*. Springer, 2011, pp. 329–332.
[3] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "A Checkpoint-on-Failure protocol for algorithm-based recovery in standard MPI," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 477–488.
[4] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in mpi," *Computing*, vol. 95, no. 12, pp. 1171–1184, 2013.
[5] "MPI Forum Standardization Effort," http://meetings.mpi-form.org.
[6] M. Ali and P. Strazdins, "Application level fault recovery: Using fault-tolerant open mpi in a pde solver," in *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW14), PDSEC*, 2014.
[7] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, and B. R. de Supinski, "Evaluating user-level fault tolerance for mpi applications," in *Proceedings of the 21st European MPI Users' Group Meeting*. ACM, 2014, p. 57.
[8] Todd Gamblin, "MPI Resilience," https://github.com/tgamblin/mpi-resilience.
[9] F. H. Streitz, J. N. Glosli, M. V. Patel, B. Chan, R. K. Yates, B. R. de Supinski, J. Sexton, and J. A. Gunnels, "Simulating solidification in metals at high pressure: The drive to petascale computing," in *Journal of Physics: Conference Series*, vol. 46, no. 1. IOP Publishing, 2006, p. 254.
[10] J. N. Glosli, D. F. Richards, K. Caspersen, R. Rudd, J. A. Gunnels, and F. H. Streitz, "Extending stability beyond cpu millennium: a micron-scale atomistic simulation of kelvin-helmholtz instability," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 2007, p. 58.
[11] X. He and L.-S. Luo, "Theory of the lattice boltzmann method: From the boltzmann equation to the lattice boltzmann equation," *Phys. Rev. E*, vol. 56, pp. 6811–6817, Dec 1997. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.56.6811

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.